Final Project Report

Sean Folan & Kevin Dao

December 2024

1 Project Overview

For this project, we went with implementing 2 different algorithms that has not been covered in class, and evaluate them two existing domains. The algorithms that we went with were model-based RL algorithms, which are Monte Carlo Tree Search and Prioritized Sweeping in particular. The domains that we are choosing to test our algorithms on are the Cat vs Monsters domain, which was introduced in Homework 3, and the 687-Gridworld domain, which was introduced in class. To go above and beyond, we stress test our algorithm on the most challenging domain yet, which is named the "Extra Large Gridworld", featuring a 10 by 10 gridworld, with 2 unique "monster" tiles, and an additional 4 sets of actions which are "AUL", "AUR", "ADL", and "ADR"; or simply diagonal actions. Visualization and details are attached at the end of the report for easy reference.

2 Monte Carlo Tree Search

2.1 Overview

Monte Carlo Tree Search (MCTS) is an online algorithm that attempts to estimate the q function of a state though random simulation. Everytime it visits a state, it builds an ExpectiMax search tree incrementally. The search can be terminated after a given amount of time or an amount of expanded nodes. Typically more useful for huge state spaces, famously used in AlphaGo (Go) and Pluribus (No limit Texas Hold'em Poker).

2.2 Algorithm

The Algorithm has four parts, which are repeated until the computational budget is met.

1. **Selection** - Select an unexpanded node.
2. **Expansion** - If we are not in a terminal state, we expand one or more of the children nodes
3. **Simulation** - Choose one of the new nodes and perform Monte Carlo simulation of the MDP
4. **Backpropagation** - The return is backpropagated up to the root

Steps one and two are defined by a TreePolicy which tells the algorithm how to select and expand and step three utilizes DefaultPolicy which encodes how the simulations are carried out.

2.2.1 Upper Confidence Trees

One of the most popular tree policies is Upper Confidence Trees (UCT). This strategy views the selection process as a multi armed bandit problem and helps us balance exploration and exploitation in the Select phase. We will use this as our TreePolicy and tune C_p for the best results.

$$\arg\max_{a\in A} Q(s,a) + 2C_p\sqrt{\frac{2\ln N(s)}{N(s,a)}}$$

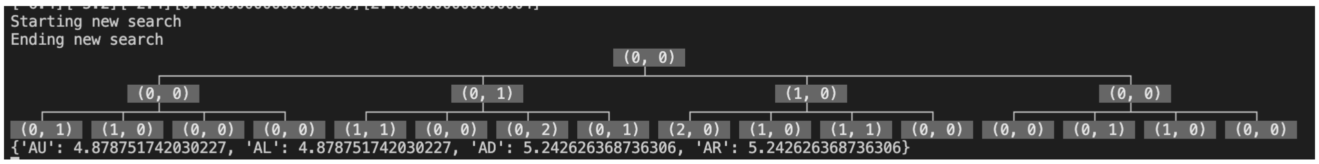
2.3 Pseudocode

Algorithm 1 MCTS

```
Input: MDP $M = (\mathcal{S}, \mathcal{A}, p, d_0, R, \gamma)$, Time limit $T$, current state $s_0$
Output: Estimated $Q$ function
while $time < T$ do
    node ← Select($s_0$)                ▷ Find a node that is not fully explored
    child ← Expand(node)                ▷ Expand the node to get the node you will start the Simulation from
    $G$ ← Simulate(child)               ▷ Run the episode getting return $G$
    Backpropagate( node, $G$)           ▷ Return results all the way up to the parent node
end while
```

2.4 Really Simple Gridworld Example

I first tested my MCTS algorithm on a domain called ”Really_Simple_Gridworld” which was a gridworld with deterministic transitions. This made it simple to try different hyperparameters and see their effects in a easy to understand deterministic domain. Below is an example tree generated in this domain. We start at $(0, 0)$ and keep selecting, expanding, simulating and backpropagating our results until we reach our rollout limit (in this case 20). Then we get the values for $Q((0, 0), \cdot)$ displayed below the tree. We observe that both ”AD” and ”AR” have the highest estimated q-value so we pick one, take the action and repeat the search from that node.



```
Starting new search
Ending new search
                                                  (0, 0)
          (0, 0)                  (0, 1)                  (1, 0)                  (0, 0)
(0, 1) (1, 0) (0, 0) (0, 0) (1, 1) (0, 0) (0, 2) (0, 1) (2, 0) (1, 0) (1, 1) (0, 0) (0, 0) (0, 1) (1, 0) (0, 0)
{'AU': 4.878751742030227, 'AL': 4.878751742030227, 'AD': 5.242626368736306, 'AR': 5.242626368736306}
```

Figure 1: A Tree made MCTS starting at state (0,0) in Really_Simple_Gridworld

2.5 Hyperparameter Search

- Exploration Factor C_p

  The typical value for the Exploration Factor is $\sqrt{2}$ but I decided to try values between 1 and 5. In general the higher the exploration parameter is, the more the algorithm will explore in the Select stage rather than Exploit.

- Default Policy

  Default Policy is the way that that outcomes are simulated in the Simulate step of MCTS. It is the policy we use to estimate the reward of our newly expanded node. A better Default policy allows for better estimates to be made of a state’s reward. I tried three general types of policies Random, Majority AR/AD and State Score which gave each state a score according to various metrics (proximity to bad states, good states, good ”areas”, etc.) and prioritized going to higher scored states.

  - Random

    Pick an action randomly until you reach the goal state.

  - Majority AR/AD

    Pick a general direction and go there most of the time. For example, pick AU/AR 35% of the time and AL/AU 15% of the time.

- State Score

  The idea that works the best is giving states a score by some function. Changed depending on the domain

Hyperparameter Graphs for each domain are shown in 3

2.6 Learning Curve

For MCTS learning curves, I compared the amount of rollouts a tree had (how many times it expanded) to the total reward. Graphs are included in 3

2.7 Graphs

2.7.1 687-Gridworld

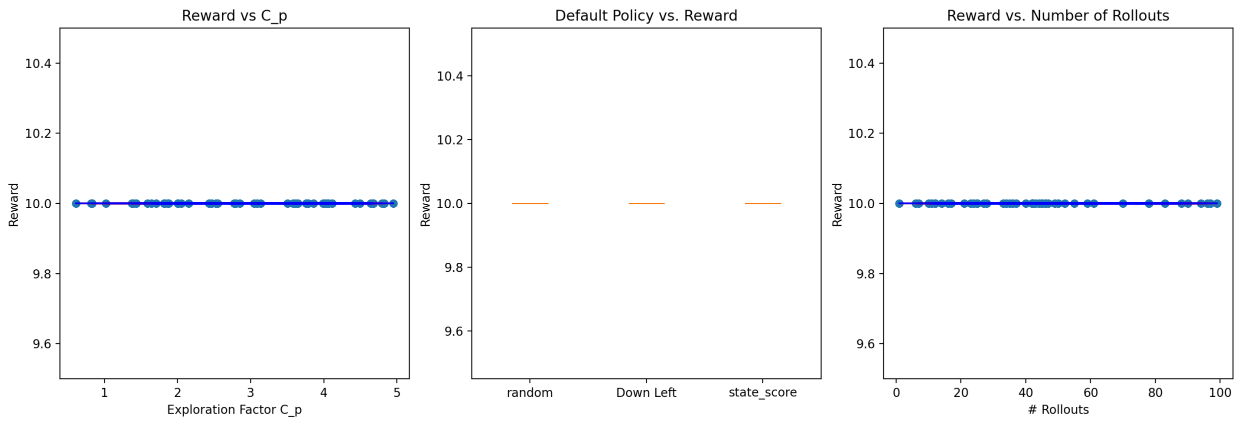


Figure 2: Gridworld Graphs

The simplest domain yielded the simplest results. With only one possible way to lose points, you almost have to go out of your way to not get 10. In the left most graph, we see that C_p does not have much of an effect because even if the trees do end up exploring a lot or a little, they are pretty unlikely to decide to take a branch that leads to a state with negative reward.

In the middle, we a box and whiskers plot of the reward earned by a total random Default Policy, a "Down Left" policy (pick AU/AR 35% of the time and AL/AU 15% of the time) and a state_score policy which assigns a score of `manhattan_dist_to_monster - 2 * manhattan_dist_to_goal`[1] and then subtracts 2 if it is a self-transition. Though all of these have similar scores, what the graphs don't show is the time it took to run each. The random policy was tediously slow and sometimes couldn't even find the goal (it hit the cutoff of 500 simulated steps to reach the goal). The Down Left Policy was fast and the state_score even faster than that.

Finally, more of the same for the number of rollouts. The algorithm can do pretty good even if it doesn't have that big of a tree. It isn't displayed in this graph but pretty much the only way MCTS can get a negative reward (or one less than ten) is with a really small number of rollouts (1 or 2). Then it sometimes happens upon the -10 square.

The hyperparameters I came up with for this domain were $\sqrt{2}$ for C_p, state_score for Default Policy, and 50 for rollouts. This seemed to perform well on time and of course produced the expected reward

[1]Any variable such as `manhattan_dist_to_goal` is manhattan distance to the nearest instance of the object, goal, monster, princess etc.

2.7.2 Cats vs Monsters

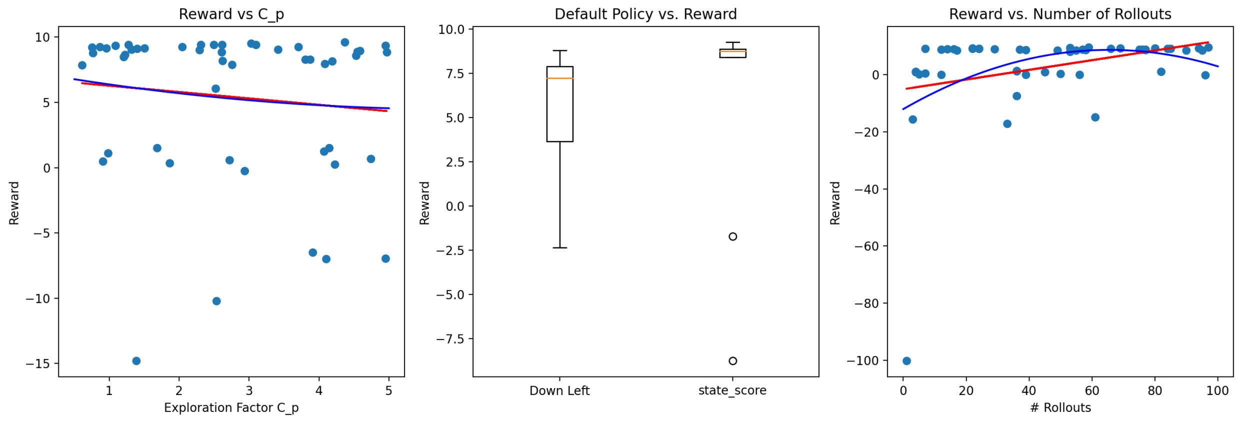


Figure 3: Cats vs Monsters Graphs

Now we get to a domain that is a little more complicated. For the impact of C_p on the reward, our line of best fit shows a slight decrease as C_p goes up. It seems like pretty much no matter what C_p is within this range, the agent usually gets a reward around 10 but sometimes runs into a monster once or twice.

For the Default Policy, we can see that the state_score policy far out does the Down Left Policy with the median being very close to ten and only a few outliers getting negative numbers. In this policy, state_score is calculated as `0.8 * manhattan_dist_to_monster - 2 * manhattan_dist_to_goal` with a -2 penalty for self transitioning. The Random policy in this case was too sub optimal to even consider.

Our number of rollouts vs reward shows in general that more rollouts is better. Our quadratic best fit function suggests that the sweet spot is around 50 but it is close even as we reach 100. When picking this hyperparameter, I factored in time and performace. I could have picked a tree size big enough to map out the whole state space but this would be inefficient and only improve marginally from the performance we can already get around 50.

The hyperparameters I picked for this domain were $C_p = 3$, state_score and rollouts = 50.

2.7.3 Extra Large Gridworld

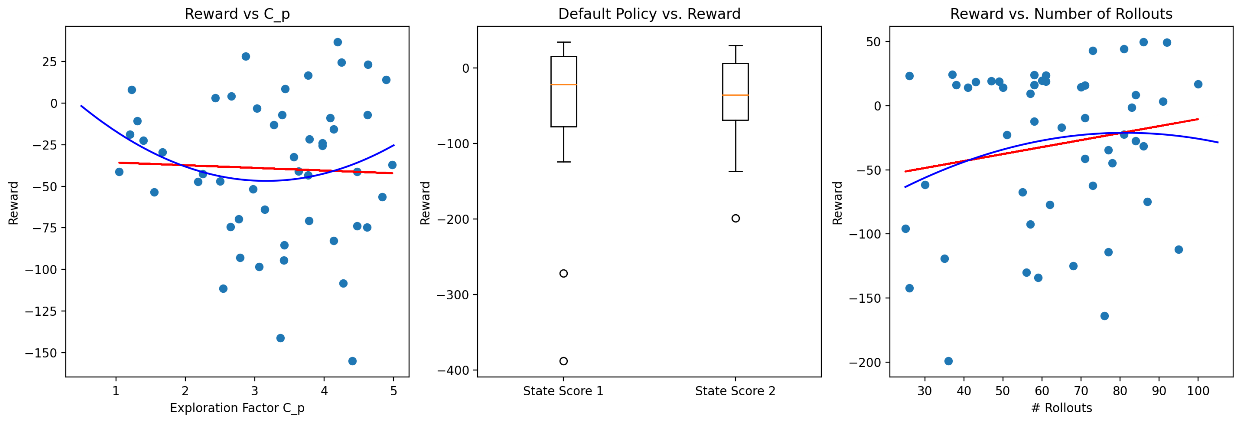


Figure 4: Extra Large Gridworld Graphs

The size and different mechanics of this domain made it harder for the algorithm. For C_p, the reqrd seemed to stay more or less constant but there seemed to be a slight downwards trend in the average reward as we get closer to 5. It seems that the rewards become more widespread with more exploration which makes sense.

For the default policy, I designed two different State Scores. State Score 1 was similar to the state score methods described above with the score being calculated as `10 * manhattan_dist_to_trap + 20 * manhattan_dist_to_guard - 50 * manhattan_dist_to_goal + 75 * inside_the_castle`[2] with a reward of -50 for self transitioning. This was working but I noticed the algorithm sometimes tending to stay in and around the top left corner. In an attempt to disuade this, I made state_score2 which is the same as state_score1 with an added reward of -75 for being "close to start" ($0 <= x <= 2$, $0 <= y <= 2$). When I tested this, state score 2 seemed to be faster but the box plot shows that state score 1 does achieve slightly higher rewards.

For the rollouts vs reward we se‘e a similar pattern to the above algorithm where there is generally a higher reward for more rollouts, but the reward only increases slightly and the cost is the time it takes the algorithm to complete.

For hyperparameters for this algorithm, I chose $C_p = \sqrt{2}$, Default Policy = state score 1 (though State Score 2 could be used if you wanted faster time with a slight decrease in performance) and 75 for rollouts.

2.8 Sources

1. https://gibberblot.github.io/rl-notes/single-agent/mcts.html
2. http://incompleteideas.net/book/RLbook2020.pdf
3. http://www.incompleteideas.net/609%20dropbox/other%20readings%20and%20resources/MCTS-survey.pdf
4. https://courses.cs.washington.edu/courses/cse473/11au/slides/cse473au11-adversarial-search.pdf

3 Prioritized Sweeping

3.1 Overview

Prioritized Sweeping, as it name suggests, allows efficient planning in reinforcement learning by focusing on state-action pairs that are expected to yield the greatest "impact". Impact here refers to the expected magnitude of change in the value function in response to new information. Because of the propagated effects on the value estimates of predecessor states, if the value of a state changes significantly, it may cause changes in the value estimates of other states that transition into it. Unlike uniform updates, Prioritized Sweeping prioritizes updates based on "urgency" through the use of priority queue, enabling fast propagation of value changes and allowing efficient use of computational resources. Prioritized Sweeping excels at maze solving because of its consideration of priority, however, it works fine under other domain, which it will be tested on three different gridworld domains.

[2] inside_the_castle refers to a state in the middle of the grid $3 <= x <= 6$, $3 <= y <= 6$

3.2 Pseudocode

```
Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all $s, a$, and $PQueue$ to empty
Loop forever:
    (a) $S \leftarrow$ current (nonterminal) state
    (b) $A \leftarrow policy(S, Q)$
    (c) Take action $A$; observe resultant reward, $R$, and state, $S'$
    (d) $Model(S, A) \leftarrow R, S'$
    (e) $P \leftarrow |R + \gamma \max_a Q(S', a) - Q(S, A)|$.
    (f) if $P > \theta$, then insert $S, A$ into $PQueue$ with priority $P$
    (g) Loop repeat $n$ times, while $PQueue$ is not empty:
            $S, A \leftarrow first(PQueue)$
            $R, S' \leftarrow Model(S, A)$
            $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
            Loop for all $\bar{S}, \bar{A}$ predicted to lead to $S$:
                $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
                $P \leftarrow |\bar{R} + \gamma \max_a Q(S, a) - Q(\bar{S}, \bar{A})|$.
                if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority $P$
```

Prioritized Sweeping is different from algorithms that has been covered in class because of its use of a ”Model” function, which stores the reward and next state explored given a state and an action, as well as a Priority Queue, a well known property of search algorithms such as Dijkstra’s.

To get the action A using the policy on state S and Q-value function Q, I opted for a soft-max approach that allows for continuous small exploration.

Prioritized Sweeping computes the priority using a modified TD-learning approach where instead of using state value function for its computation, it uses state-action value function with the value maximizing action. If the priority exceed a threshold theta, then it is inserted into the priority queue.

The Q-value function update considers the first ”n” elements of the priority queue, updating the corresponding Q-value function for state-action pair using the already computed priority and a step size alpha. Changes are propagated to parent state-action pairs, where its priority is computed and considered to add into the priority queue.

3.3 Hyperparameters Fine-tuning

There are 6 hyperparameters that I am fine-tuning for, those are:

- theta (θ) : priority threshold
- alpha (α) : step-size
- epsilon (ϵ) : exploration rate
- n : number of q value update
- niter : number of episodes
- episode length : number of steps per episode

Leveraging the hyperparameter search that we have done in Homework 2, I used Evolution Strategy to find the best set of parameters for each particular domain, Cat vs Monsters and 687-Gridworld.

There are 4 main parts in any Evolution Strategy which are population generation, fitness evaluation, mutation, and off-spring selection

For population generation, I drew a uniform distribution of continuous variables such as theta, alpha and epsilon and drew random integers for discrete variables such as n, niter, and episode length. Because I want my evolution strategy to finish in reasonable time and exploiting the fact that most hyperparameters have bound, I set bounds for each hyperparameter, where theta, alpha, and epsilon ranges from 0.01 to 0.75, n ranges from 1 to 100, and niter and episode length ranges from 50 to 500. I drew a total of 20 set of hyperparameters.

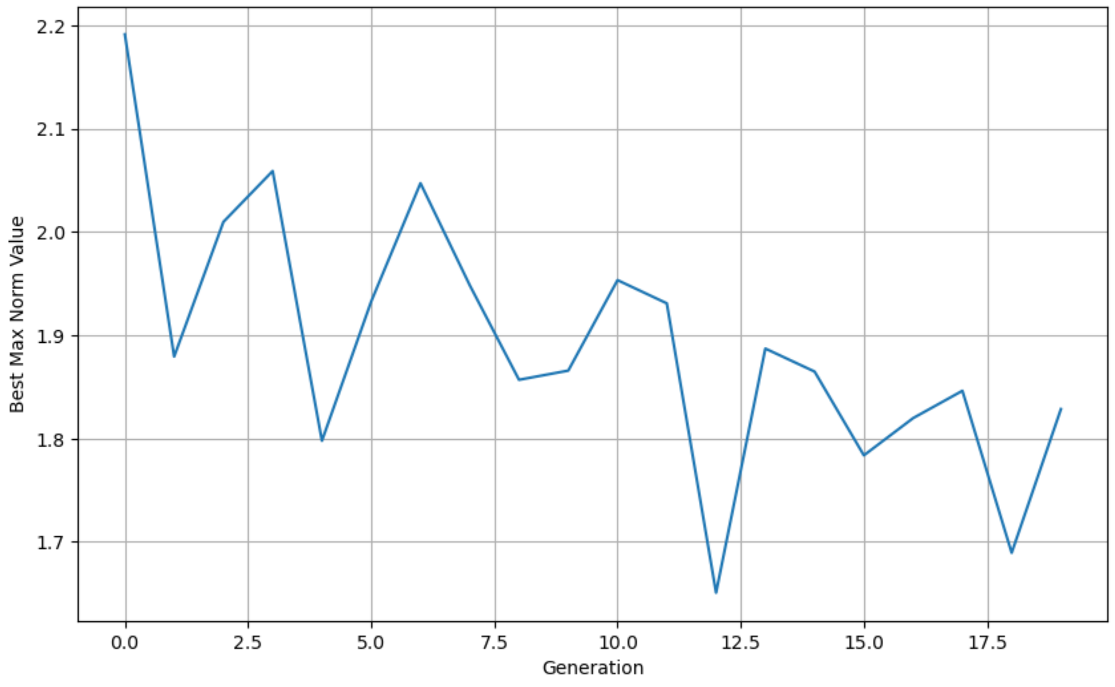
For fitness evaluation, I used the max-norm value of the state value function of interest (converted from the outputted state-action value function, which is derived from a set of hyperparameters in the population of the current generation) and the optimal value function (gotten from running Value Iteration). Because of the stochastic nature of these domains, I averaged the max-norm value across 10 runs for each set of hyperparameters to get a lower bias fitness value.

For mutation, I sampled a normal distribution where the mean is 0 and the standard deviation is the difference in the bound for the respective hyperparameter times a constant "mutation strength". For discrete variables, I changed my mutation slightly, where the noise is rounded and divided by 10 to prevent explosion of hyperparameters, especially for large bounds. The mutation strength decays

For off-spring selection, I evaluated each parent with the fitness evaluation as discussed above. The evaluation will return a "score", which is the Max-Norm Value when running Prioritized Sweeping on the "parent" hyperparameters. After the algorithm finishes evaluating, I choose the top 5 parents to seed my next generation's off spring, where 2 of the lowest loss parents will be part of the next population to increase the robustness of the evolution process. This evolution process repeats for 20 generations.

3.4 Learning Curves

Cat vs Monsters' Learning Curve

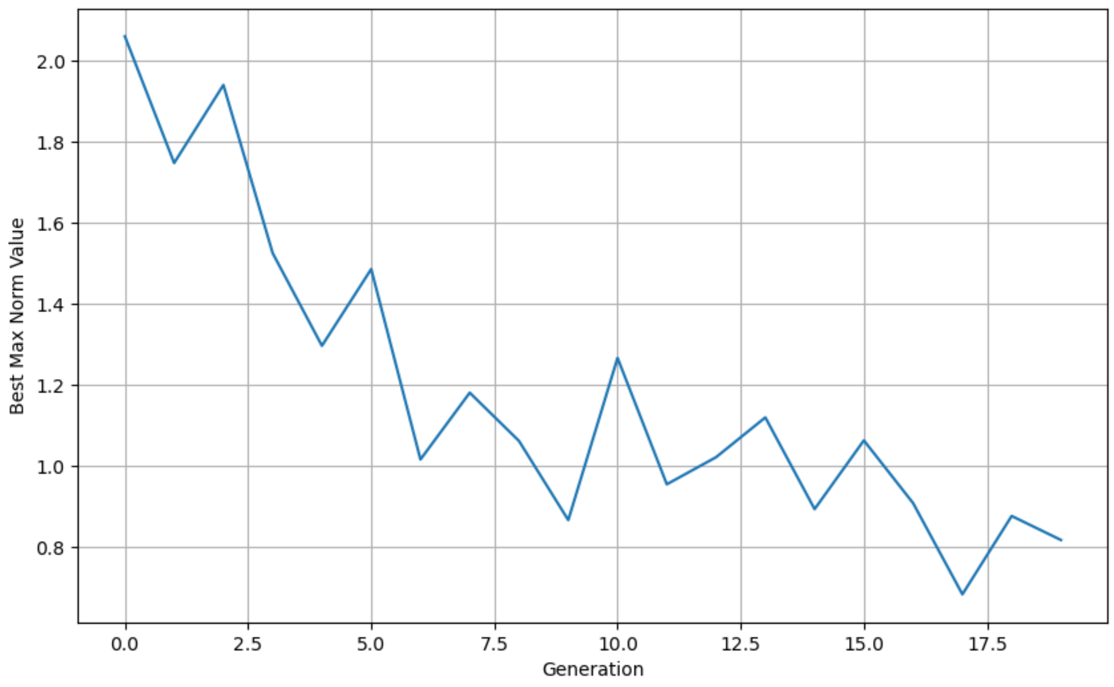


The final parameters found were:

- $\theta = 0.8384$
- $\alpha = 0.0735$
- $\epsilon = 1.0$
- $n = 8$
- niter $= 329$
- episode length $= 370$

The best Max-Norm Value was 0.7518

687-Gridworld's Learning Curve



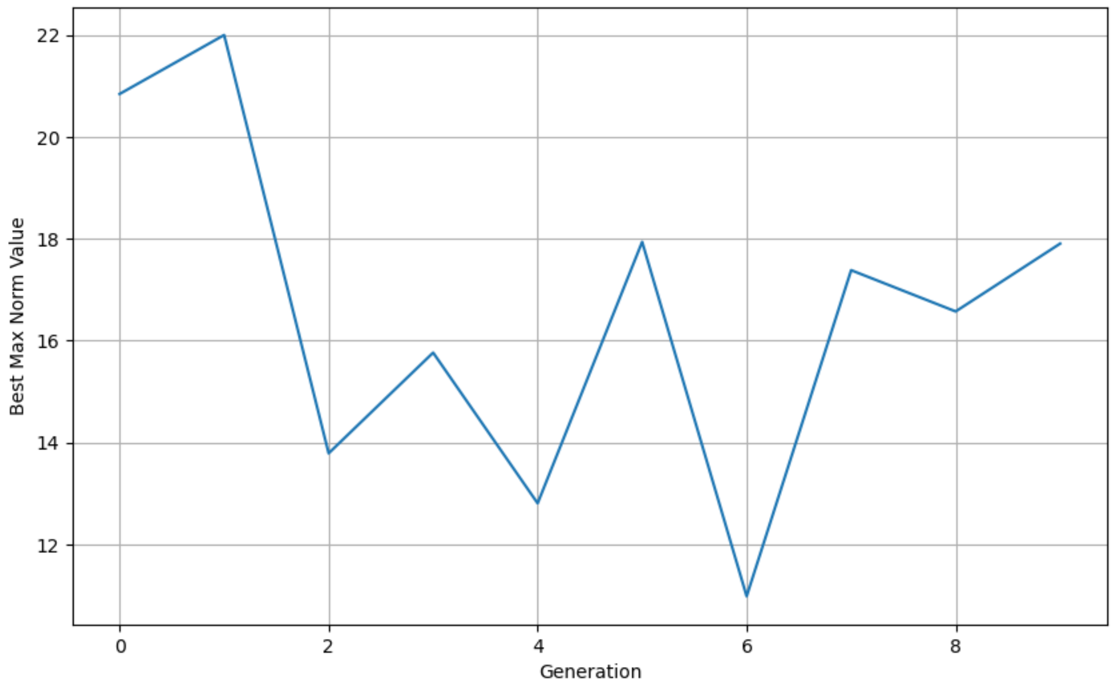
The final parameters found were:

- $\theta = 0.8367$
- $\alpha = 0.8425$
- $\epsilon = 1.0$
- $n = 52$
- niter $= 249$

- episode length = 168

The best Max-Norm Value was 0.2038

Extra-Large-Gridworld's Learning Curve



The final parameters found were:

- $\theta = 0.2073$
- $\alpha = 0.0185$
- $\epsilon = 0.1665$
- $n = 72$
- niter = 69
- episode length = 103

The best Max-Norm Value was 10.9797

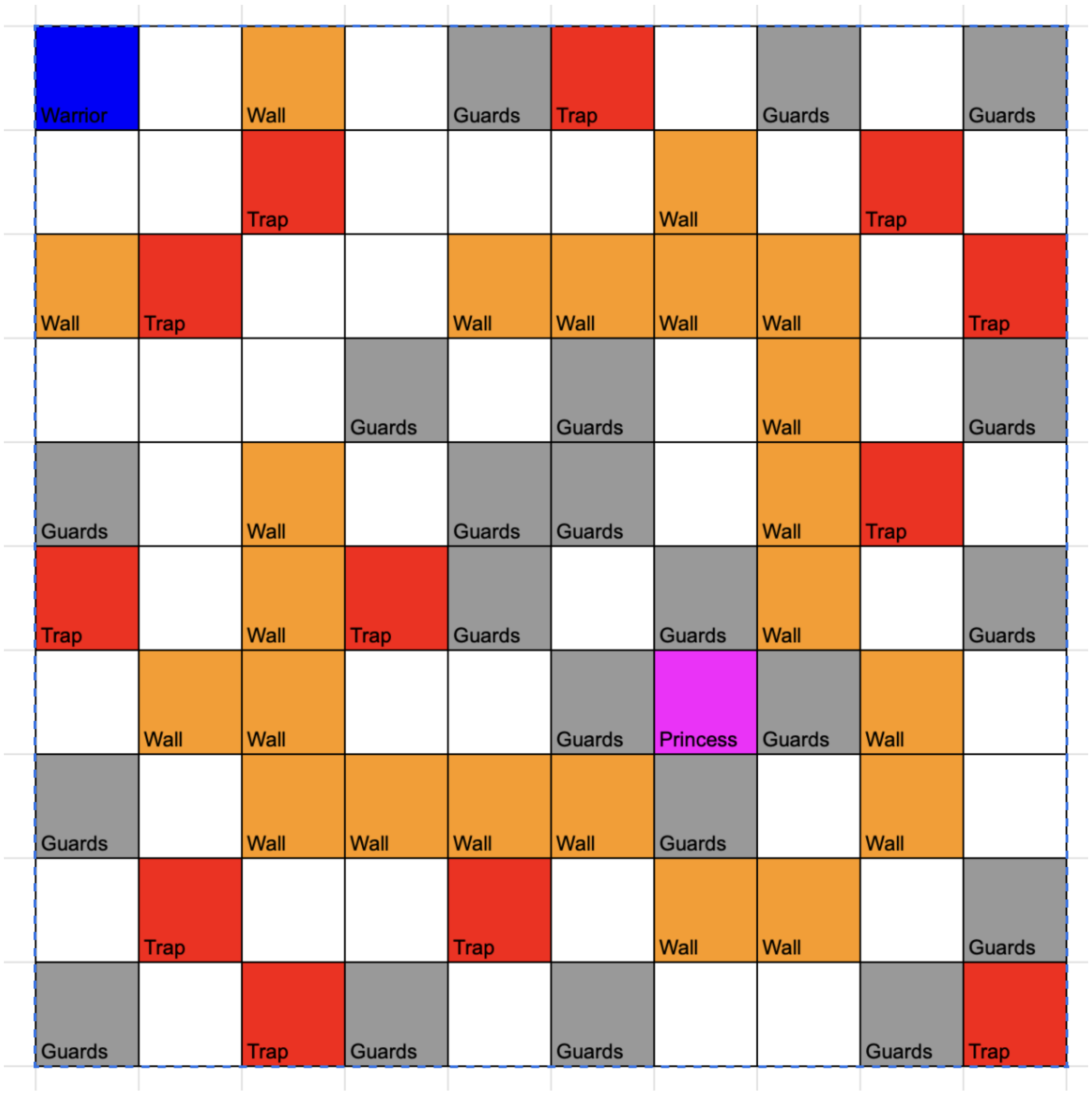
Comment:

I had multiple trial runs with testing out different bounds for each hyperparameters, the number of generations and populations, that makes the most sense for the computing resource I have available with me. Like I have indicated above, I decided on an Evolution Strategy variance that draws a population of 20 across 20 generations, where the fitness evaluation is averaged across 10 runs. I choose the top 5 parents in each generation to seed the next generation, where the top 2 parents is directly fed into the next generation to reduce evaluation variance across generations. The mutation strength, which is the degree in which the offspring varies from the parent was set to 0.05, where the mutation strength is multiplied by the difference of the bounds for said hyperparameter, and that value is the standard deviation where the next generation is drawn from.

I landed on these numbers because drawing a population of 20 across 20 generations made the most sense to me in terms of computing resource. Running the Evolution Strategy under this configuration and where the fitness function is evaluated across 10 runs takes around 30 minutes for each domain. In my earlier runs, I did not consider averaging the fitness score across multiple runs, making the graph varied greatly across multiple runs and can therefore not be reliably claimed as fine-tuned. I first experimented with only the highest scoring parent can seed the next generation, but again, that makes the search algorithm extremely susceptible to outliers. By creating a seeding logic as described above, I could clearly see a clear trend of the evolution strategy algorithm learning a progressively better set of hyperparameters.

4 Reference

4.1 Visualization of Extra-Large-Gridworld domain



Where white cells are "empty" cells. The blue cell is the "cat" of this game and is also the starting state.

The reward scheme is similar to Cat vs Monsters, where transitioning to an 'empty' cell nets a reward of -0.05, 'trap' nets -5, 'guard' nets -20, and 'princess' nets 50 and is also the terminal state.